LA-UR -81-1550

CONF-810523--2

TITLE: EVENT-ANALYSIS LANGUAGE FOR HIGH-SPEED DATA ACQUISTION

AUTHOR(S): R. A. Hardekopf, R. V. Poore, J. W. Sunier, M. Neiman, and A. Holm

SUBMITTED TO: Conference on Computerized Data Acquisition, Oak Ridge, TN, May 1981

MASTER

DISCLAIMER

University of California

LOS ALAMOS SCIENTIFIC LABORATORY
Post Office Box 1663 Los Alamos, New Mexico 87545
An Affirmative Action/Equal Opportunity Employer

EVENT-ANALYSIS LANGUAGE FOR HIGH-SPEED DATA ACQUISITION*

R. A. Hardekopf, R. V. Poore, J. W. Sunier

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

M. Neiman

Lawrence Berkeley Laboratory
Berkeley, Ca. 94720

A. Holm

Niels Bohr Institute
Copenhagen, Denmark

ABSTRACT

We describe a multiparameter data-acquisition program that is highly flexible and yet provides optimum machine code for each individual set of sorting conditions. An Event Analysis Language (EVAL) compiler allows the user full control over the handling of events and produces code which runs 4-6 times faster than a generalized program. We have integrated this technique into the Los Alamos Physics Division data acquisition system for high-speed sorting of a wide variety of input data from CAMAC or magnetic tape. The EVAL compiler is written in FORTRAN for MODCOMP computers, but can be easily modified for other systems.

I. INTRODUCTION

EVAL is an event analysis program used for data acquisition and for sorting event tapes. The basic idea exploited by EVAL is that sorting multiparameter data consists of many repetitions of a single sorting algorithm. If a language could be defined to easily describe such algorithms, a compiler could be written which would produce optimum machine code for each individual set of sorting conditions. This allows the user full control over the handling of the events and produces code which runs much faster than a generalized program. EVAL was originally developed[1] for sorting events from magnetic tape. We describe here its extension to on-line data acquisition.

One way to consider EVAL is as a program which transforms the computer into a programmable calculator with a large memory. Each multiparameter event is presented to the calculator individually. The user must write a program, in the EVAL language, which specifies how each event is to be treated. The calculator has an accumulator which can contain a number (either integer or floating point). It can test the value in the accumulator, do arithmetic with it, and use it as a channel number of a spectrum in memory to be incremented. While it can do both integer and floating point arithmetic, integer arithmetic is, of course, much faster.

About 30 commands, with suitable arguments, are sufficient to handle the manipulations required in most types of data acquisition. This includes masking and shifting the raw events, conditional branching based on gating conditions, performing arithmetic or bit manipulation with the events, incrementing spectra, and writing the events to tape. In addition, a subroutine capability allows the user to invoke any FORTRAN or assembly language code for more specialized requirements.

II. EXAMPLE

Before providing details on the EVAL language, it is instructive to consider an example that, although relatively simple, illustrates many of its features. The following is a complete EVAL procedure that is given the name GSORT, followed by a discussion of the program.

In reading the program the following points should be noted:

1) While a name may be any length, only the first four characters are significant.

2) The delimiters space, comma, and equals are equivalent and may be used interchangeably to improve readability.

3) C in column 1 denotes a comment line. Comments may also be put on each EVAL command line by preceding the comment with colon (:).

4) The line numbers are added to aid in the discussion of the program. They are not a part of the program itself.

5) The first 2 lines (without numbers) are not part of the EVAL program. In the command structure[2] of the Los Alamos Physics Division program "Z", the first line defines a procedure called GSORT which exists as a directoried text file on disc. The second line loads the EVAL compiler, which then interprets lines 1-44.

```
     PROC GSORT
     EVA
1.   BRA
2.   C  DATA STRUCTURE DEFINITIONS
3    FORMAT    GE1    1    12    1
4    FORMAT    GE2    2    12    1
5.   FORMAT    TAC    3    12    3
6.   SPEC      S1     1
7.   SPEC      S2     2
8    SPEC      ST1    3
9    SPEC      ST2    4
10.  SPEC      SR1    5
11.  SPEC      SR2    6
12.  GATE      G1     1     1    1    1
13.  GATE      G2     1     1    1    2
```

```
14.  GATE      TRUE    1     7     1     1
15.  GATE      RANDOM  1     7     1     2
16.  DATA      EVSIZE=3
17.  C  START OF EXECUTABLE CODE DEFINITIONS
18.  TAPE
19.  GET X=GE1
20.  INC S1
21.  GET Y=GE2
22.  INC S2
23.  GET TAC
24.  BRA
25.  IF TRUE
26.            BRA
27.            IF X G1
28.                      INC Y ST1
29.            ELSE
30.            IF G2
31.                      INC Y ST2
32.            KET
33.  ELSE
34.  IF RAND
35.            BRA
36.            IF X G1
37.                      INC Y SR1
38.            ELSE
39.            IF G2
40.                      INC Y SR2
41.            KET
42.  KET
43.  KET
44.  C END OF EVAL Compilation
```

In the above example, a 3 parameter experiment is sorted. In lines 3-5 the three parameters are given names and declared to be in words 1, 2, and 3 of the event. The ADC's produce 12 bits (4096 channels) which are placed in bits 12 to 1 of each word. The least significant bit is bit 1 and the most significant bit is bit 16. Therefore, the declaration of line 3 includes all 4096 channels while line 5 drops the 2 least significant bits resulting in a 1024 channel parameter. These FORMAT declarations will be used to compile code for masking and shifting the events when they are brought into the accumulator with a GET statement.

In lines 6-11 the 6 spectra to be sorted are declared. Spectra S1 and S2 will be used for singles. The others are coincidence spectra. The numbers refer to data areas that have been previously defined by the user; the EVAL compiler gets the information that it needs (e.g., type, length, and address) from a memory allocation file.

In lines 12-15 some gates are declared. The numbers refer to spectrum set, spectrum, gate set and gate buttons on the display button panel[3], and define various gates to be used for sorting. In the Los Alamos "Z" system, these gates are intensified regions of the spectrum display and can be changed interactively. In line 16 the constant EVSIZE is set to 3, meaning all events are three words long. If events are not all the same size, EVSIZE can be set as a variable.

Lines 18-42 contain the executable part of the program. Line 18 has the TAPE command and indicates that at this point the event is to be placed in a tape buffer. The buffer will automatically be written to tape when it is full. In line 19 the parameter described by the format GE1 is both loaded into the accumulator and stored in the variable X. In line 20 the spectrum S1 is incremented in the channel contained in the accumulator. Thus a singles spectrum is created. Lines 21-22 do a similar thing for GE2.

In line 23 the TAC is loaded into the accumulator. Line 24 is the first half of a BRA-KET which ends on line 42. BRA-KETS, together with IF and ELSE are used to control program flow. A group of statements enclosed by a BRA-KET pair is seen as one indivisible statement from anywhere in the program outside of the BRA-KET. From inside of the BRA-KET one can only jump to another point inside the BRA-KET. Line 25 contains an IF statement which tests the accumulator against the gate TRUE. If the accumulator contains a number within the current setting of the TRUE gate, the IF statement is satisfied and the program proceeds to line 27. If not, it jumps to the next visible ELSE or to the end of the BRA-KET, in this case to the ELSE on line 33.

Lines 26-32 contain a BRA-KET which will be executed only if the IF statement on line 25 is satisfied. On line 27 the value X is brought into the accumulator and then tested against gate G1. If in the gate, line 28 is executed which fetches Y into the accumulator and then increments spectrum ST1. Control would then jump to the end of the BRA-KET (line 32). If the gate in line 27 is not true, the program would jump to the ELSE on line 29 and continue by executing line 30. On line 30 the accumulator is compared to gate G2. If it is in the gate, line 31 is executed which will bring Y into the accumulator and increment spectrum ST2; otherwise the program will jump to the end of the BRA-KET (line 32). When the ELSE on line 33 is reached, control will then jump to the end of that BRA-KET (line 42) and the treatment of that event will be terminated.

Line 34 contains an IF statement which can only be reached in the case where the IF statement on line 25 is not satisfied. Lines 35-41 contain a BRA-KET similar in structure to the BRA-KET in lines 26-32.

To sum up, the illustrated sorting program has looked at a GELI-GELI coincidence. It has produced 6 spectra. Two GELI singles (lines 19-22) and 4 coincidence spectra. There are 2 coincidence spectra of GELI-2 with gates on GELI-1 for both a TRUE and RANDOM gate on the TAC. In this example, the TAC spectrum is not being stored, but it would be simple to add this with only 2 commands; e.g., a SPEC T command and INC T after line 23. In practice, one would need to store this spectrum during setup in order to set the gates, but the INC could then be removed for faster sorting.

III. THE EVAL LANGUAGE

All EVAL statements are one line long and begin with a predefined code word. Following the code word are possibly some parameters. The notation is:

A) < >indicates one parameter of the type described by the word in the < >, i.e., <NAME> means a text name.

B) A vertical line | is read as "or". For example, <DATA|VARIABLE> means either data or a variable is needed.

C) [ ] means an optional parameter. For example, some commands that operate on the accumulator also allow a variable to be loaded into from the accumulator within that command.

D) NUM refers to a constant, FORMAT is the name of a word in an event, SPEC is the name of a 1 or 2 dimensional spectrum and DATA refers to a constant defined by a DATA specification statement.

The code words are divided into 5 main groups and are listed below. The allowed parameters are also indicated.

1) DECLARATIONS:

DATA <NAME> <VALUE>

SPEC <NAME> <DATA AREA NUMBER>

FORMAT <NAME> <WORD><MS>BIT><LS BIT>

GATE <NAME> <SS SP GS G CS>

VARIABLE<NAME> <VALUE>

OPTION <NAME> [<NAME, NAME....>]

2) DATA AND SPECTRUM MANIPULATION:

GET [<VAR>] <FORMAT>

INC [<VAR>] <SPEC>

TINC [<VAR>] <SPEC>

LDA <NUM|DATA|SPEC|VAR>

STA <VAR|SPEC|FORMAT>

SUB <NUM|DATA|FORMAT>

ADD <NUM|DATA|FORMAT|SPEC>

MUL <NUM|DATA|FORMAT>

DIV <NUM|DATA|FORMAT>

CHS

FIX

FLOAT

INDEX [<NUM|DATA|VAR>]

3) BIT MANIPULATION:

LSH <INTEGER|DATA>

ASH <INTEGER|DATA>

OR <INTEGER|DATA|VAR|FORMAT>

AND <INTEGER|DATA|VAR|FORMAT>

XOR <INTEGER|DATA|VAR|FORMAT>

4) PROGRAM FLOW CONTROL:

BRA

RET

IF [<VAR>] <GATE|LOGICAL TEST>

ELSE

MARK

SUB1 [<VAR>]

5) WRITING THE EVENTS TO TAPE:

TAPE [<FORMAT|VAR>]

The functions of the code words which were not used in the example are:

OPTION: Allows certain options to be in effect during compile time. For example, NOEV prevents the normal event counter (for dead-time corrections) from being added to the code, and SAMP creates a variable used for sampling only part of the data buffers.

VARIABLE: Defines an EVAL variable and gives it an initial value.

TINC: Is the same as INC except the channel number in the accumulator is tested against the size of the spectrum before the spectrum channel is incremented.

LDA: Load the accumulator.

STA: Store the accumulator.

SUB,ADD,MUL,DIV: Perform those operations on the accumulator, leaving the result in the accumulator.

CHS: Change the sign of the accumulator.

FIX, FLOAT: Fix or float the accumulator.

INDEX: Transfer a value from the accumulator to the index register. The index register is used for examining 2-dimensional gates and for indexing into a spectrum.

LSH: Logical shift on accumulator.

ASH: Arithmetic shift on accumulator.

OR: Logical OR to the accumulator.

AND: Logical AND to the accumulator.

XOR: Exclusive OR to the accumulator.

MARK: Denote the beginning of an event loop in the EVAL program. Can be used, for example, with the TAPE ALL and OPTION SAMP commands to transfer the entire data buffer to the tape buffer while histogramming only a sample.

SUB1: Name of a FORTRAN or assembly language subroutine to be called. Up to 8 user written subroutines may be used (SUB1.....SUB8).

The load accumulator (LDA), store accumulator (STA) and ADD commands also allow operations on spectra. In these cases the channel number to be loaded into the accumulator, stored into from the accumulator or added to the accumulator must be in the index register. This option allows, among other things, the capability of multiscaling.

Variables (VAR) are names of memory locations and are declared implicity by their use in statements or by the VARIABLE statement. Variables defined in an EVAL code may be changed "on the fly" by a "Z" command. FVSIZE must be included in each program, either as a DATA constant or as a VARIABLE. It must be set equal to the size of each event in 16 bit words. At the end of the EVAL program the event pointer is incremented by EVSIZE and the program is entered again at the top if there are more events in the buffer.

The entire EVAL program must be contained in a BRA, KET pair, i.e. the first line of the program must be BRA and the last line must be KET. Upon encountering the final KET the compiler completes the code, installs it at the proper location in memory, and exits.

COMPILER DETAILS

The EVAL compiler is a FORTRAN code that contains assembly-language instructions as hexadecimal op-codes in a data statement. The normal mnemonic for the op-code is used whenever possible. Memory is allocated on a 256-word page basis as needed, and instructions and locations are loaded into this area as the compiler interprets EVAL commands. A global common defined by the program contains the page numbers of the code as well as spectrum and gate descriptions and variables defined by the commands. The commands are normally read from a directoried text file on disc which can be included in a user's setup procedure or can be a stand-alone file. A macro processor capability with arguments allows duplicating sections within the EVAL code with minimum effort.

A local symbol table keeps track of each item name, kind, and value as commands are encountered. The available command mnemonics are also stored in this table by the program. Thus if text is present in a command, the program first looks to see if an operation, spectrum, gate or variable by that name has been previously defined. If not, and the operation is allowed, the unknown text is assumed to be a new variable and this is inserted in the table. Appropriate error messages are generated if required.

The MODCOMP 15 general purpose registers are used in the sorting code as follows:

1 spare
2 spare
3 index register used with 2-D gates and the LDA, STA and ADD commands
4 base address of the current event
5 spare
6 first address of the EVAL variables
7 scratch
8 scratch
9 scratch
10 return address in calling program
11 last address of input event buffer
12 extra accumulator extension
13 extra accumulator
14 accumulator extension
15 accumulator

The program keeps track of the accumulator status (integer, real, or undefined) and the index register status during the compilation. Some commands can automatically fix or float the accumulator as necessary for proper operation, while others give error messages for an improper sequence.

Once the sorting program is completed and the EVAL compiler exits, tasks are activated by appropriate "Z" commands for data acquisition or tape reading. Up to 8 EVAL tasks can run simultaneously for each user, each independently sorting and storing multiparameter data by branching to the code compiled for that task. The tasks are resumed by interrupt when the respective input data buffers are filled. When the data is activated, the buffers are filled by DMA transfer using lists of CAMAC commands stored in the Differential Branch Driver[4] interface. When storing event data on tape, the tape buffers are filled automatically by the sorting tasks and written when full. The user need not be concerned with these details, but needs only to construct the sorting instructions for each event.

If a change in the sorting algorithm is required, the user can easily make necessary changes in the EVAL source text and execute the EVA command to compile the new version. Upon completion, the EVAL compiler automatically installs the new machine language code in the sorting task and data acquisition can begin again. If frequent changes in the sorting algorithm are required, EVAL variables can be inserted in the code and conditional branching on these variables produces alternative paths. The value of EVAL variables can be interrogated or changed at any time by appropriate "Z" commands.

The external subroutine capability is not as easy to implement, but can be extremely powerful for certain applications. When the EVAL compiler encounters a SUBn command, code is generated to save the active registers and branch to the subroutine through a table in the common area. If variables are to be passed as arguments, their number and addresses are also generated. The actual address for the subroutine is put into the common by the sorting task when it is loaded. In order to implement a new subroutine, the sorting task must be re-compiled and linked with the custom routine. A simple job-control procedure exists in the system for accomplishing this.

V. APPLICATIONS AND RESULTS

EVAL was implemented at Los Alamos only about 9 months ago, but is already used in many different applications. Many of the users have converted from the general multiparameter code MUL[3] in order to take advantage of the increased sorting speed and flexibility offered by EVAL. For these cases, including multiple particle-telescopes, neutron time-of-flight, GELI coincidence, and wire-chamber proportional counters, the sorting speeds are roughly 4-6 times faster. EVAL is more efficient primarily because the sorting algorithm is "compiled" into the code rather than interpreting a large data structure to get the sorting instructions.

The EVAL flexibility has also allowed users to sort their data in ways that are difficult to implement in a general code, and with little or no help from a systems programmer. For example, the execution of floating point instructions to do gain stabilization, and conditional branching prior to putting events in the tape buffer to name only two. This freedom in control of the input events, however, must be tempered by user caution since they can lose all of their data by improper event handling. Usually, such errors are easily detected on compilation or checkout. The implementation of EVAL has also greatly reduced the work load on our systems programmers, since most users can now write their own sorting programs. This factor alone results in increased productivity for the computer staff.

In addition to the MODCOMP data acquisition systems at Los Alamos, the EVAL compiler has been implemented on a Digital VAX-11/780 with some minor modifications, described by a paper in these proceedings.[5] Another version is being developed at Los Alamos to compile macrocode instructions for a Bulk Memory Processor[6] where word lengths of 24 bits and memory up to 16 megawords will be available for multiparameter sorting.

REFERENCES

*Supported by the U.S. Department of Energy.

1. A. Holm, "Construction of Efficient Nuclear Event Analysis Programs Using a Simple Dedicated Language", IEEE Transactions on Nuclear Science NS-26, No. 4, 4569 (1979).

2. R. V. Poore, D. E. McMillan, R. O. Nelson and J. W. Sunier, "Data Acquisition System at Los Alamos P-9 Accelerator Facility", IEEE Transactions on Nuclear Science, NS-26 No. 1, 708 (1979).

3. J. W. Sunier, R. V. Poore, and D. E. McMillan, "A Multiparamter Data Acquisition and Display Program", IEEE Transactions on Nuclear Science, NS-26, No. 4, 4485 (1979).

4. D. E. McMillan, R. O. Nelson, R. V. Poore, J. W. Sunier and J. J. Ross, "A High Speed CAMAC Differential Branch Highway Driver", IEEE Transactions on Nuclear Science, NS-26, No. 4, 4450 (1979).

5. L. G. Holzweig and R. V. Poore, "Event Analysis Language EVAL for VAX-11/780", Proceedings of this conference.

6. R. O. Nelson, D. E. McMillan, J. W. Sunier, M. Meier and R. V. Poore, "Bulk Memory Processor for Data Acquisition", Proceedings of this conference.